****

**DD2459 Software Reliability**

**Take Home Examination 2020**

|  |  |
| --- | --- |
| **Name** | **Bryan Leow Xuan Zhen** |
| **Number of Pages (excluding cover page)** | **9** |
| **T-number** | **960107T377** |
| **Personal Mobile Number** | **0728707109** |
| **E-mail addresses** | [**bryanlxz@kth.se**](mailto:bryanlxz@kth.se)  [**blxz96@gmail.com**](mailto:blxz96@gmail.com)  [**xleow002@e.ntu.edu.sg**](mailto:xleow002@e.ntu.edu.sg) |

**Question 1**

|  |  |
| --- | --- |
| **(i)** |  |
| **(ii)** |  |

|  |  |
| --- | --- |
| **(iii)** |  |
| **(iv)** | From the precondition, we can see that:  *myCharSet* can take any non-null array with unique characters as inputs for test case formulation.  *myChar* can take any character as inputs for test case formulation.  For pairwise test case, we first need to set the defaults for *myCharSet* and *myChar.*  Since pairwise testing includes 0-wise testing, 1-wise testing and 2-wise testing, our test suite must contain the following:  a default test case, a test case whereby only *myCharSet* is taking non-default value, a test case whereby only *myChar* is taking non-default value and a test case whereby both *myCharSet* and *myChar* is taking non-default values.  Then, we evaluate the postcondition after execution of each test case to ensure that should there exist an index *i* in *\old* (*myCharSet)* where *\old* (*myCharSet[i])* == *myChar,* then *myCharSet* is unmodified and *\result* == *myCharSet == \old (myCharSet)*. Else, *myCharSet* will be modified to take in *myChar* and the length of *myCharSet* will be incremented by 1. Subsequently, *\result* == *myCharSet* *&& myCharSet! = \old (myCharSet).* |
| **(v)** | Best way to look at the dimensionality of system under test is to see *myCharSet* as one dimension and *myChar* as another dimension. As mentioned earlier, *myCharSet* can take any non-null array with unique character. Should we be considering the possible combination for *myCharSet,* there are infinite possibilities. Hence, the best way is to just consider *myCharSet* as a variable and myChar as another. Hence, n = 2. |

**Question 2**

|  |  |
| --- | --- |
| **(a)** | **G (saveMoney -> F(rich))** |
| **(b)** | **G (brake -> X2(stop) | X3(stop) -> (stop ⋃ !brake))** |
| **(c)** | **G ( green | red ) &**  **G (! ( green & red ) ) &**  **red &**  **G ( red & press -> X(press) -> X2(green) & green & press -> X(press) -> X2(red) )** |
| **(d)** | |  |  |  | | --- | --- | --- | | **Test Case** | **Test Case Values** | **Expected Values** | | **TC1** | **red & press -> X(press)** | **X2(green)** | | **TC2** | **green & press -> X(press)** | **X2(red)** | | The aspect of (c) that we cannot test is the time taken between the current state in which the button is pressed for the first time and the next state in which the button is pressed for the second time. | | | |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **(e)** | |  |  | | --- | --- | | **State Machine Diagram** |  | | “*If tomorrow is Sunday then the day before yesterday was Wednesday*” | **G (X(Sunday) -> X5 (Wednesday))** |   According to the false PTPL temporal formula, **G (X(Sunday) -> X5 (Wednesday)).**  **Counterexample:**  From the state machine diagram, assume X(Sunday).  Then, **X(Sunday) -> X2 (Monday) -> X3 (Tuesday) -> X4 (Wednesday) -> X5 (Thursday).**  Therefore, **!G (X(Sunday) -> X5 (Wednesday)).** |

**Question 3**

|  |  |
| --- | --- |
| **(a)** |  |
| **(b)** | There are 13 edge requirements in ER.  **ER\_TR1: n1, n2**  **ER\_TR2: n2, n3**  **ER\_TR3: n3, n4**  **ER\_TR4: n3, n5**  **ER\_TR5: n4, n6**  **ER\_TR6: n5, n6**  **ER\_TR7: n6, n7**  **ER\_TR8: n6, n8**  **ER\_TR9: n7, n2**  **ER\_TR10: n8, n2**  **ER\_TR11: n2, n9**  **ER\_TR12: n9, n10**  **ER\_TR13: n9, n11** |
| **(c)** | There are 2 requirements in RER.  **RER\_TR1: n1, n2, n3, n4, n6, n7, n2, n9, n10**  **RER\_TR2: n1, n2, n3, n5, n6, n8, n2, n9, n11**  Covered by both **RER TR1** and **TR2** **:** **ER TR1, TR2** and **TR11**  *Covered by only* ***RER\_TR1 : ER TR3, TR5, TR7, TR9*** *and* ***TR12***  *Covered by only* ***RER\_TR2 : ER TR4, TR6, TR8, TR10*** *and* ***TR13***  Since there are TR in ER that is covered only by RER\_TR1 or RER\_TR2, the smallest subset of RER has a size of 2. |

|  |  |
| --- | --- |
| **(d)** | The goal of structural testing is to exercise a minimum collection of locations. “Enough” testing is defined in terms of coverage rather than test suite size in structural testing. Hence, RER is a better requirement set for testing than ER because it yields the same edge coverage with a smaller test suite size (2 < 13) and this will help to save a lot of time, allowing the coverage to be more easily and accurately defined. |
| **(e)** | |  |  |  |  |  | | --- | --- | --- | --- | --- | | **Test Case** | **Test Case Values** | **Setup Values** | **Teardown Values** | **Expected values** | | **TC1** | **x = -1 , y = -2** | **x’= x, y’= y** | **x’= x, y’= y** | **-2 (since return y ⬄ return -2 )** | | **TC2** | **x = 2 , y = 1** | **x’= x, y’= y** | **x’= x, y’= y** | **2 (since return x ⬄ return 2 )** | |
| **(f)** | There are 16 test requirements for E2R.  **E2R\_TR1: n1, n2, n3**  **E2R\_TR2: n1, n2, n9**  **E2R\_TR3: n2, n3, n4**  **E2R\_TR4: n2, n3, n5**  **E2R\_TR5: n3, n4, n6**  **E2R\_TR6: n3, n5, n6**  **E2R\_TR7: n4, n6, n7**  **E2R\_TR8: n4, n6, n8**  **E2R\_TR9: n5, n6, n7**  **E2R\_TR10: n5, n6, n8**  **E2R\_TR11: n6, n7, n2**  **E2R\_TR12: n6, n8, n2**  **E2R\_TR13: n7, n2, n9**  **E2R\_TR14: n8, n2, n9**  **E2R\_TR15: n2, n9, n10**  **E2R\_TR16: n2, n9, n11** |

|  |  |
| --- | --- |
| **(g)** | There are 5 test requirements for RE2R.  **RE2R\_TR1: n1, n2, n3, n4, n6, n7, n2, n9, n10**  **RE2R\_TR2: n1, n2, n3, n4, n6, n8, n2, n9, n11**  **RE2R\_TR3: n1, n2, n3, n5, n6, n7, n2, n9, n10**  **RE2R\_TR4: n1, n2, n3, n5, n6, n8, n2, n9, n11**  **RE2R\_TR5: n1, n2, n9, n10**  Covered by **RE2R TR 1, 2, 3, 4 : E2R TR1**  *Covered by only* ***RE2R TR 5 : E2R TR2***  Covered by both **RE2R TR 1, 2 : E2R TR3, TR5**  Covered by both **RE2R TR 3, 4 : E2R TR4, TR6**  *Covered by only* ***RE2R TR 1 : E2R TR7***  *Covered by only* ***RE2R TR 2 : E2R TR8***  *Covered by only* ***RE2R TR 3 : E2R TR9***  *Covered by only* ***RE2R TR 4 : E2R TR10***  Covered by both **RE2R TR 1, 3 : E2R TR11, TR13**  Covered by both **RE2R TR 2, 4 : E2R TR12, TR14, TR16**  Covered by **RE2R TR 1, 3, 5 : E2R TR15**  Since there are TR in E2R that is covered only by RE2R TR1, TR2, TR3, TR4 or TR5 alone, the smallest subset of RE2R has a size of 5. |
| **(h)** | RE2R is more effective than RER because it allows more coverage in nature. From the requirement, we can see that RER only allows us to check the case where x’ and y’ are both negative or are both greater than or equal to zero. However, RE2R allows us to check both or them are positive or negative or only one of them is positive and another negative. By virtue of this, RE2R is more effective than RER. |
| **(i)** | The mathematical function is defined as |
| **(j)** | |  |  |  |  |  | | --- | --- | --- | --- | --- | | **Test Case** | **Test Case Values** | **Setup Values** | **Teardown Values** | **Expected values** | | **TC1** | **x = -1 , y = -2** | **x’= x, y’= y** | **x’= x, y’= y** | **-2 (since )** | | **TC2** | **x = 2 , y = 1** | **x’= x, y’= y** | **x’= x, y’= y** | **2 (since )** | |